PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

A1

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04L 29/06 | A2

(11) International Publication Number: WO 99/48261

(43) International Publication Date: 23 September 1999 (23.09.99)

(21) International Application Number: PCT/US99/05991

(22) International Filing Date: 18 March 1999 (18.03.99)

(30) Priority Data:

09/040,832	18 March 1998 (18.03.98)	US
09/040,827	18 March 1998 (18.03.98)	US

(71) Applicant: SECURE COMPUTING CORPORATION [US/US]; 2675 Long Lake Road, Roseville, MN 55113 (US).

(72) Inventors: REID, Irving; 152 Saint Patrick Street #610, Toronto, Ontario (CA). MINEAR, Spencer; 1291 Gardena Avenue, Fridley, MN 55432 (US). FLINT, Andrew; 29 Bridgewater Road, Oakville, Ontario (CA). AMDUR, Gene; 135 George Street South #704, Toronto, Ontario (CA).

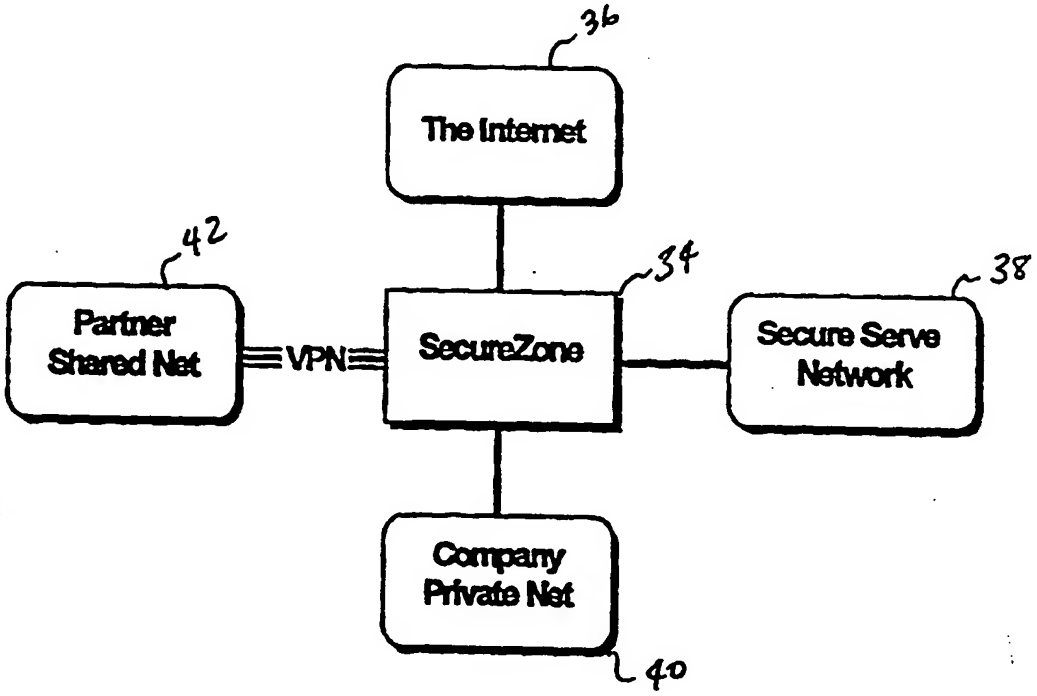
(74) Agent: HOLLOWAY, Sheryl, S.; Schwegman, Lundberg, Woessner & Kluth, P.O. Box 2938, Minneapolis, MN 55402 (US).

(81) Designated States: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published

Without international search report and to be republished upon receipt of that report.

(54) Title: SYSTEM AND METHOD FOR CONTROLLING INTERACTIONS BETWEEN NETWORKS



(57) Abstract

A firewall is used to achieve network separation within a computing system having a plurality of network interfaces. A plurality of regions is defined within the firewall and a set of policies is configured for each of the plurality of regions. The firewall restricts communication to and from each of the plurality of network interfaces in accordance with the set of policies configured for the one of the plurality of regions to which the one of the plurality of network interfaces has been assigned.

1A

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

SYSTEM AND METHOD FOR CONTROLLING INTERACTIONS BETWEEN NETWORKS

Field of the Invention

The present invention relates generally to network security, and more particularly to a system and method of grouping networks to enforce a security policy.

Background of the Invention

Recent developments in technology have made access easier to publicly available computer networks, such as the Internet. Organizations are increasingly turning to external networks such as the Internet to foster communication between employees, suppliers and clients. With this increased access comes an increased vulnerability to malicious activities on the part of both people inside and outside the organization. Firewalls have become a key tool in controlling the flow of data between internal networks and these external networks.

A firewall is a system which enforces a security policy on communication traffic entering and leaving an internal network. Firewalls are generally developed on one or more of three models: the screening router, the bastion host, and the dual homed gateway. These models are described in U.S. Patent No. 5,623,601 to Vu, issued April 22, 1997 and entitled APPARATUS AND METHOD FOR PROVIDING A SECURE GATEWAY FOR COMMUNICATION AND DATA EXCHANGES BETWEEN NETWORKS (Vu), which is hereby incorporated herein by reference.

Vu describes packet filters as a more sophisticated type of screening that operates on the protocol level. Packet filters are generally host-based applications which permit certain communications over predefined ports. Packet filters may have associated rule bases and operate on the principle of that which is not expressly permitted is prohibited. Public networks such as the Internet operate in TCP/IP protocol. A UNIX operating system running TCP/IP has a capacity of 64K communication ports. It is therefore generally considered

impractical to construct and maintain a comprehensive rule base for a packet filter application. Besides, packet filtering is implemented using the simple Internet Protocol (IP) packet filtering mechanisms which are not regarded as being robust enough to permit the implementation of an adequate level of protection. The principal drawback of packet filters, according to Vu, is that they are executed by the operating system kernel and there is a limited capacity at that level to perform screening functions. As noted above, protocols may be piggybacked to either bypass or fool packet filtering mechanisms and may permit skilled intruders to access the private network.

Accordingly, it is an object of this invention is to provide a method for controlling interactions between networks by the use of firewalls with defined regions.

Summary of the Invention

The present invention is directed to a system and method of achieving network separation within a computing system having a plurality of network interfaces. One aspect of the invention is a method comprising the steps of defining a plurality of regions; configuring a set of policies for each of the plurality of regions; assigning each of the plurality of network interfaces to only one of the plurality of regions, wherein at least one of the plurality of network interfaces is assigned to a particular region; and restricting communication to and from each of the plurality of network interfaces in accordance with the set of policies configured for the one of the plurality of regions to which the one of the plurality of network interfaces has been assigned.

Another aspect of the invention is a secure server comprising an operating system kernel; a plurality of network interfaces which communicate with the operating system kernel; and a firewall comprising a plurality of regions, wherein a set of policies have been configured for each of the plurality of regions; wherein each of the plurality of network interfaces is assigned to only one of the plurality of regions; wherein at least one of the plurality of network interfaces is assigned to a particular region; and wherein communication to and from each of the plurality of network interfaces is restricted in accordance with the set of policies configured for the one of the plurality of regions to which the one of the plurality of network interfaces has been assigned.

A feature of the present invention is the application level approach to security enforcement, wherein type enforcement is integral to the operating system. Still another feature is protection against attacks including intruders into the computer system. Yet another feature is a new graphical user interface (GUI) in effective Access Control Language (ACL). A further feature of the present invention is a visual access control system. Another feature is embedded support for Virtual Private Networking (VPN).

Brief Description of the Drawings

Figure 1 depicts an implementation of the firewall of the present invention.

Figure 1a shows a representative computing system protected by a firewall.

Figure 1b depicts another computing system protected by a firewall.

Figure 2 shows the regions and their members as defined in the present invention.

Figure 3 is a graphical representation of ACL commands.

Figure 4 is a flow diagram for a virus alert.

Figure 5 depicts a method by which incoming data packets are processed in accordance with the present invention.

Detailed Description of the Preferred Embodiments

In the following detailed description of the preferred embodiments, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

Figure 1 depicts a block diagram showing the relationship between a firewall 34 in accordance with this invention, the Internet 36, a Secure Server Network (SSN) 38, a Company Private Net 40, and a Partner Shared Net 42. As shown in Figure 1, communications to and from any other servers or networks goes through the firewall 34.

Two representative firewall-protected computing systems are shown in Figures 1a and 1b. System 10 in Figure 1a includes an internal network 12 connected through firewall 14 to external network 16. A server 18 and one or more workstations 20 are connected to internal network 12 and

communicate through firewall 14 with servers or workstations on external network 16.

System 30 in Figure 1b includes an internal network 32 connected through firewall 34 to external network 36. A server 38 and one or more workstations 40 are connected to internal network 32. In addition, a server 42 is connected through network 44 to firewall 34. Workstations 40 communicate through firewall 14 with servers or workstations on external network 16 and with server 42 on network 44. In one embodiment network 44 and server 42 are in a sort of demilitarized zone (DMZ) providing protected access to server 42 to internal users and to external entities.

In one embodiment, firewalls 14 and 34 implement a region-based security system as will be discussed below.

Regions are a new and flexible way of organizing systems such as systems 10 and 30. Regions let you group both physical interfaces (network cards) and Virtual Private Networks (VPNs) into areas of similar trust and security needs. Regions (along with services) provide the foundation on which every access rule is built. By grouping together networks and VPNs that require the same type of security, you eliminate the need to enter multiple versions of the same access rule for each network or VPN. In doing so, regions give you the flexibility to tailor a security policy that meets the specific needs of your network environment.

One embodiment of a region-based system is shown in Figure 1. In Fig. 1, firewall 34 coordinates communication between internal network 32 (e.g., a company private network), external network 36 (e.g., the Internet) and DMZ network 44 (e.g., a secure server network). In one such embodiment, firewall 34 also controls virtual private network (VPN) communication between external entities and networks 32 and 44. Regions are defined and one or more networks is assigned to each region. In the example shown in Figure 3, the regions are Sales Office, Worldwide Customer Service, Worldwide Sales, Secure 'DMZ' and R&D Network. R&D Network includes the trusted internal network. Sales Office and Secure 'DMZ' are within slightly less trusted regions. Worldwide Customer Service and Worldwide Sales come in unencrypted over the Internet and are, therefore, the least trusted.

Firewall 34 protects regions from unauthorized access through the use of access rules. For each connection attempt, the Firewall checks it against the defined access rules. The rule that matches the characteristics of the connection request is used to determine whether the connection should be allowed or denied.

The operating system on which the firewall 34 is implemented is the BSDI 3.1 version of UNIX, a security hardened operating system with each application separated out, and protected by type enforcement technology. The functions of firewall 34 are all integrated with the operating system, and each one is completely compartmentalized and secured on its own, and then bound by type enforcement control.

Type enforcement, which is implemented within the operating system itself, assures a very high level of security by dividing the entire firewall into domains and file types. Domains are restricted environments for applications, such as FTP and Telnet. A domain is set up to handle one kind of application only, and that application runs solely in its own domain. File types are named groups of files and subdirectories. A type can include any number of files, but each file on the system belongs to only one type.

There is no concept of a root super-user with overall control. Type enforcement is based on the security principle of least privilege: any program executing on the system is given only the resources and privileges it needs to accomplish its tasks. On the firewall of this invention, type enforcement enforces the least privilege concept by controlling all the interactions between domains and file types. Domains must have *explicit permission* to access specific file types, communicate with other domains, or access system functions. Any attempts to the contrary fail as if the files did not exist. The type enforcement policy is mandatory, and nothing short of shutting the system down and recompiling the type enforcement policy database can change it.

Type enforcement is described in two pending patent applications entitled SYSTEM AND METHOD FOR PROVIDING SECURE INTERNETWORK SERVICES, Serial No. 08/322,078, filed October 12, 1994,

and SYSTEM AND METHOD FOR ACHIEVING NETWORK SEPARATION, Serial No. 08/599,232, filed February 9, 1996.

Essentially, a type enforcement scheme provides for the secure transfer of data between a workstation connected to a private network and a remote computer connected to an unsecured network. A secure computer is inserted into the private network to serve as the gateway to the unsecured network and a client subsystem is added to the workstation in order to control the transfer of data from the workstation to the secure computer. The secure computer includes a private network interface connected to the private network, an unsecured network interface connected to the unsecured network, wherein the unsecured network interface includes means for encrypting data to be transferred from the first workstation to the remote computer, a server function for transferring data between the private network interface and the unsecured network interface and a filter function for filtering data transferred between the remote computer and the workstation.

The firewall of the present invention features application-level gateways, which negotiate communications and never make a direct connection between two different networks. Hence, unlike packet filtering, which, as described in the prior art, applies rules on every incoming packet of data, the firewall applies rules applicable to the network or port in which data packets are entering. The gateways have a detailed understanding of the networking services they manage. This architecture isolates activity between network interfaces by shutting off all direct communication between them. Instead, application data is transferred in a sanitized form, between the opposite sides of the gateway.

In addition to the firewall's secured type enforced operating system and application gateway architecture, the system has been designed to defend against known network penetration and denial of service attacks.

Finding out who and where attacks are originating from is a key requirement to taking corrective action. The firewall also includes intruder response that allows administrators to obtain all the information available about a potential intruder. If an attack is detected or an alarm is triggered, the intruder response mechanism collects information on the attacker, their source, and the route they are using to reach the system.

In addition to real-time response via pager or SNMP, alarms can be configured to automatically print results or to email them to the designated person.

The growing need for applying specific security policies and access requirements to complex organizations requires a new way of managing firewalls – regions. Regions are groupings of physical interfaces (network cards) and virtual networks (VPNs) into entities of similar trust.

Suppose a company has thousands of roaming users connecting to the company network from encrypted virtual private network ("VPN") clients – managing such users one at a time would be an enormous task. It would be easier to organize those roaming users into groups having, as an example, full access, medium access, and limited access rights. Figure 2 depicts regions Internet, Secure 'DMZ', R&D Network, Sales Offices, Worldwide Customer Service, and Worldwide Sales. In Figure 2, all Sales or Customer Support departments in the company's offices can be grouped together into regions Worldwide Sales and Worldwide Customer Service, respectively.

Regions permit the grouping of networks and VPNs that require the same type of security, thereby eliminating the need to enter multiple versions of the same access rule for each network or VPN. Thus regions allow flexibility in tailoring a security policy. In defining regions, the first task is to group together networks or VPNs that require the same type of network access. Each network interface card or VPN that is grouped in a region is considered a member of that region. A region can consist of the following members:

- an interface card,
- a VPN,
- a group of VPNs,
- an interface card and a VPN, or
- an interface card and a group of VPNs.

Hence in Figure 2, user1, user2, user3, mgr1, and mgr2 of Region named R&D Network would have the same rights defined for the R&D Region. In the same way, Roaming Sales 1, Roaming Sales 2, Roaming Sales 3, etc. would have the same rights accorded to all members of Region named Sales Offices. In Figure 2, user1, user2, Roaming Sales 1, Roaming Sales 2, mgr1,

etc., do not necessarily represent only workstations. In other words, it is possible for user2 to logon the workstation onto which user3 might ordinarily logon, or for mgr1 to logon the workstation onto which mgr3 might ordinarily logon.

Every region is protected from every other region as defined in the firewall of the present invention. All connections to and from each region are first examined by the firewall. Regions may communicate with each other only if an appropriate access rule has been defined. For each access rule, first, the services that the rule will control must be defined, then, second, the regions that the connection is traveling between must also be defined. For example, if the Internal region is to be allowed to access Telnet services on the Internet region, the access rule must specify Telnet as the service that the rule controls and specify the *From:* region as Internal and the *To:* region as Internet. Hence, the firewall of the present invention does not allow traffic to pass directly through the firewall in any direction. Region to Region connections are made via an application aware gateway. Application-level gateways understand and interpret network protocol and provide increased access control ability.

The ACLs are the heart and soul of the firewall. For each connection attempt, the firewall checks the ACLs for permissions on use and for constraints for the connection. Constraints can include: encryption requirements, authentication requirements, time of day restrictions, concurrent sessions restrictions, connection redirection, address or host name restrictions, user restrictions and so forth.

Access rules are the way in which the firewall protects regions from unauthorized access. For each connection attempt, the firewall checks it against the defined access rules. The rule that matches the characteristics of the connection request is used to determine whether the connection should be allowed or denied.

With the firewall of the present invention, access rules are created in a completely new way – using decision trees. Knowing that an access rule is based on a series of decisions made about a connection, the firewall permits the building of an access rule based on "nodes" of decision criteria. A node can be added to check for such criteria as the time of day, whether the connection uses the appropriate authentication or encryption, the user or groups initiating the

connection request or the IP address or host of the connection. Each node is compared against an incoming connection request and you determine whether the connection is allowed or denied based on the results of the node comparison.

Every access rule must consist of two specific nodes. The first, the Services node, decides which service(s) the rule will control. The second, the From/To node determines the source region and destination region of the connection. Once the services and regions for the rule are established, more nodes can be added to determine specific details about the connection.

This approach provides a new way to control network access. The Firewall presents access rules as visual decision tree diagrams. Each diagram contains building blocks or nodes of information that apply a condition to or make a decision about the connection. At any point, you can add alerts to indicate when a particular point in an access rule has been reached or filters to check for authentication, encryption, WWW blocking or FTP commands.

In addition to the Allow or Deny terminal nodes, there are four other types of nodes you can add to an access rule: decision nodes, filter nodes, redirects and alerts. Decision nodes will be discussed next.

At any point in an access rule, you can check a connection request based on the time of day, its users and groups, its IP addresses and hosts or maximum concurrent sessions At these decision nodes, the Firewall determines whether the connection is true or false. If the connection meets the criteria listed in the node, the connection is considered true and proceeds along a "true" branch. If the connection does not meet the node criteria, the connection is considered false and proceeds along a "false" branch.

You can apply a filter at any point in an access rule. Filters differ from decision nodes in that they do not determine if a connection is true or false. Instead, filters attempt to apply a condition to the connection. If the filter can be applied to the connection, the filter is performed and the connection proceeds along the same path. If the filter does not apply to the connection, the filter is ignored and the connection still proceeds. In one embodiment, the filter node can force user authentication or encryption, can use filters to block particular WWW connections, or can filter the connection to see if it contains Java or ActiveX content.

A rewrite node is a point in an access rule where source or destination addresses are mapped to other source or destination addresses. Destination IP address rewrites allow an inbound connection through NAT address hiding to be remapped to a destination inside the NAT barrier. Source address rewrites can be used on outbound connections to make the source appear to be one of many external addresses. This process allows the internal hosts to be aliased to external addresses. In one embodiment, rewrites can be based on any connection criteria, including users.

At any point in an access rule, you can add an alert that notifies recipients when a connection has reached a particular point in an access rule. Using these alerts, you can monitor specific users, IP addresses and other criteria contained within a specific access rule.

When a connection request reaches a node in a rule, it is checked against the information in the node. If the connection is a filter node 72, the filter condition is either applied or ignored. Only one branch leads out of a filter node. If the node happens to be a decision node, there are two possible results. If the connection meets the criteria listed, it is considered true and follows the "true" branch of the access rule. Otherwise, the connection is considered "false" and follows the false branch.

Referring to Figure 3, the design for this feature falls almost directly out of the GUI representation. The GUI presents access rules as a decision tree with special kinds of nodes which make true or false decisions. Each decision leads to a branch which contains more nodes. Along the way, filters can be acquired. These filters are not processed by the kernel with the exception of redirects (rewrite destination address or port). In Figure 3, the time of day is checked (50). If during business hours, the user is checked (52). Certain users are allowed, so connection is allowed (54) as indicated by the check mark. However, some users (56) require a SmartFilter check (58), whereas everyone else is denied (60).

The firewall of the present invention introduces a revolutionary means to manage network access control. Traditional firewalls provide lists of access control rules, but as more rules and controls are added, these lists become unmanageable. As shown in Figure 3, the present invention presents a visual

means by which access control can be defined and easily understood through flowchart style diagrams.

The firewall's access flow diagrams allow any decision criteria to be based on any other decision, in any order. If the administrator wants to check user first, then time, then apply a specific access policy, they can. In addition, the flow diagrams are object oriented for greater power.

Access control rules on the firewall can be defined with flexibility previously unknown in the industry. This allows, for example, for different web filtering polices on a per-user basis, the ability to deny a connection if it isn't encrypted, authenticate a connection by strong token and another connection by password. Access rules can incorporate any of the following criteria:

- Source and destination Region
- Users and groups
- Source and destination addresses, networks, hosts, and domains
- Type of service (WWW, Email, Telnet, FTP, etc.)
- Time of day, Day of week
- Load balancing
- Maximum number of concurrent sessions
- Required level of encryption
- Required level of authentication (strong token, password, etc.)
- Protocol filters (WWW, FTP – see later in this section)
- SmartFilter™ URL blocking policy (see later in this section)
- Multiple external IP address connected to
- Source and destination service port and IP address rewrites

The firewall's access control diagrams include the capability of *IP address rewrites*, which allows a connection inbound through NAT address hiding to be remapped to a destination inside the NAT barrier. Also, rewrites can be used on outbound connections to make the source appear to be one of many external addresses. This allows internal hosts to be aliased to external addresses.

Rewrites can be based on any connection criteria, including users. So the administrator can have anonymous FTP connections directed to a public

access FTP server on the Secure Server Net, but remap users to their internal machines.

The firewall's access control diagrams also include the capability of sending alerts, with an administrator-defined message, based on any connection decision. Alerts can be dropped into the access flow diagrams at any point. If a connection reaches that point in the diagram, the alert is triggered. For example, in Figure 4, a check for viruses is performed on a file (70). If a virus is found, the administrator is alerted (72), and the transfer is redirected to a safe location for later inspection (74).

The ACLs consist of all the required kernel code. This is all the code that implements the rules themselves in the kernel including: build, modifying, deleting, and querying the rules. Also included are the system calls that the user level programs need to use the ACLs. The parsing of the return values, especially the filters are not part of the ACLs themselves since the filter rules are defined dynamically by the programs issuing the system calls to build the ACLs. It is the intent that the kernel be flexible enough to handle all the filter requirements without needing modifications for future enhancements.

The ACLs themselves must satisfy the requirements laid out by the GUI design. This dictates to a large degree how the rules must be implemented. Since the user has no direct access to the ACLs (rather they use the user interface), there are no ease of use concerns here except to say that the ACLs must be something the developers can work with easily. Hence, there exists a good set of tools to debug the ACLs.

Virtual Private Networking (VPN) has been embedded into the architecture of the firewall of the present invention, making it an operating characteristic of the operating system, as opposed to other firewalls which added VPN later. Every access control is available to VPN connections in exactly the same way as for physically connected networks: user controls, IP restrictions, protocol filters, address hiding, multi-homing, and more. VPN is a method of authenticating and transparently encrypting bi-directional data transmissions via the Internet. Both gateway to gateway network links as well as roaming users on VPN enabled laptops are utilizing the security and cost effectiveness of VPN

Internet encrypted communications. VPN technology is embedded in the core design of the firewall of the present invention.

There are usually 2 sockets per session, client_sock and server_sock. Each socket has two endpoints, so there can be up to four different IP addresses. Note that loc_dst_addr could be anything, if the firewall bound to a wildcard address. Here are diagrams for BFS Inbound, BFS Outbound, and the firewall of the present invention.

```
                    client_sock                          server_sock
client (cli_addr) ----------> [firewall (invention) ] ----------> (srv_addr)
server
                    (loc_dst_addr)                       (loc_src_addr)
```

The SIGWINCH signal is used to force all ACLs to be rechecked and for proxies to re-initialize themselves (for proxies that use config files). Most proxies will handle this signal themselves, but if secured did an ACL before starting a proxy, it must also do the recheck. The SIGWINCH signal will come from the backend, which will use killpg() to signal all the inetd daemons, secured processes, and their child proxies or servers. Note that the default action for SIGWINCH is ignore, so inetd did not need to be modified.

Some transient proxies use the SIGALRM internally to do idle proxy timeouts (tcpgsp, tnauthp, sqlp).

All proxies should shutdown cleanly if given a SIGTERM signal. The backend (daemond actually) uses SIGTERM to kill inetd processes when the last service has been removed. We have modified inetd to catch SIGTERM and then use killpg(SIGTERM, pgid) to kill all its children (proxies and secureds). When it starts up, inetd creates a new process group and becomes the leader, which allows it to kill all children easily.

Squid will re-open (not rotate) its logfiles if given the SIGUSR1 signal, and re-initialize itself if given SIGWINCH or SIGHUP. Note that this means squid does not do ACL rechecks, it treats it just like a SIGHUP – closes its listen sockets and waits 30 seconds for active sessions to terminate, then

re-opens listen sockets. This easy way out was chosen because squid's connections are relatively short-lived.

The following options are passed to **secured** by the backend writing them on the **inetd.conf** line:

-D te_dom	Set the TE domain of our child process to te_dom
-N service_number	the service number is required for ACL calls. **secured will pass this number on to all proxies**
-t	Specifies that secured is running a transient service (with the wait flag in inetd.conf). ACL checks are not done by secured for transient services, because the service itself must do ACL checks.
-u	Specifies that this service supports the notion of a user name, so secured should let service perform its own ACL checks. Currently only FTP, telnet and WWW support user names. Note: only needed for ftpp, because tnauthp and squid already do their own ACLs.

The following options are passed to a proxy by the **backend** writing them on the **inetd.conf** line:

-a audit_name	use 'name' in call to openlog() and for auditing
-i N	specify session idle timeout as N seconds
-I N	specify proxy idle timeout as N seconds (transient only)
-P ch	specify descriptor port, ch=S for secure, ch=L for lpr, ch=G for generic, otherwise, ch=N specify fixed port, or ch=low-high to specify a port range

The following ACL return values are passed to short-lived proxies by **secured**:

-N service_number	the same service number that secured got via backend
-c cli_rgn	set cli_region
-s srv_rgn	set srv_region
-D IP	specify the server IP address
-M IP	specify an IP address to spoof as loc_src_addr, for MAT-out
-p N	specify the server port number
-P N	specify fixed value for descriptor port
-C	spoof client-side socket (typically outbound proxies)
-S	spoof server-side socket (typically inbound proxies)

By letting the ACLs control so many settings, the inetd.conf lines are much simpler and the degree of control is much greater. For example, here are some BFS inetd.conf entries for inbound proxies:

inbound_udp_relay -e 199.71.190.101 -w 65546 -u g_udp_ir -d 192.168.128.138 -m -g 0

secured -ws 144 -wr 1 -wn 1 -l 199.71.190.121 www_X www_r_i - - -d 192.168.125.2 -m

Here are the corresponding entries for the firewall of the present invention:

secured -N 123 -D RGnx -t - ntpp -a ntpp

secured -N 456 -D RGnx -t - httpp -a httpp

The following options are only used for **debugging** purposes, some might be disabled on production systems or supported in future releases:

-n	non-transparent proxy mode - only works for VDO-Live

-U user_name	set the user name (ftp ftp_mux and ftpp/ftpd)
-A ch	set the audit method, ch=s for syslog, ch=a for audit, ch=e for stderr
-m	disable socket mating
-L	disable connection logging
-z	set non-paranoid mode, which relaxes IP address checks for UDP proxies

The firewall of the present invention uses new structured audit calls for session logging, which include src and dst region, ACL matched, auth method, encryption state, etc. The new calls are:

- audit_session_begin
- audit_session_continue
- audit_session_end
- audit_log_ftp - to log FTP file transfers, includes user, filename, size
- audit_log_smartfilter - to log URL, action (allow/deny), blocked categories
- audit_acl_deny - to log ACL denials
- audit_ipsec_fail - to log IPSEC failures
- audit_auth_fail - to log authentication failures

The present firewall has incorporated the proxy-warder-interface (pwif) from Sidewinder. We also support external authentication servers such as snk, safeword, securid. The pwif interface was already supported by tnauthp, we added pwif support to ftpp, and for GUI login. We are not using pwif for squid, instead we are using their build-in passwd file support. The backend will have to

keep the squid passwd file in sync with the static-passwd file used for ftp and telnet.

Besides a simple allow/deny, the ACLs also return the following: from_region, to_region, destination redirects for IP and port, source redirects for IP and port, transparency settings and filters. We have standardized ACL filters as follows (example from acl_util.h):

```
#define FILT_DELIM      '|'
/*
 * all filters will be at least 3-characters in length
 * proxy ACL filters will all start with "p"
 * all filters should be disabled (0) by unless ACLs enable them
 */
/* generic proxy filters - all start with "pg"
 *
 * filt_debug          filter "pgdN" sets debug level to N
 *
 * filt_crypto_from    filter "pgeR:levels" requires encryption in regions R,
 * filt_crypto_to      where R equals F, T, B for from_rgn, to_rgn, both,
 * filt_crypto_levels  and levels is colon delimited in
"rc4-40:rc4-128:des56:3des"
 *                     For example, pgeF:rc4-128:3des" would force strong
 *                     encryption between the client and the firewall
 *
 * filt_loc_auth       filter "pgaX" specifies local auth
 *                     the character X gives the method: S, s, w
 *                     for STRONG_ONLY, STRONG_PREFER,
WEAK_PREFER
 *
 * filt_rem_auth       TRUE or FALSE
 * filt_undef_servers  filter "pgA:" specifies list of remote auth methods,
 *                     colon delimited "pgA:radius:safeword:securid:snk"
 */
```

```
typedef struct {
        /* generic proxy filters - see above for their defined values */
        char filt_debug;
        char filt_crypto_from;
        char filt_crypto_to;
        int  filt_crypto_levels;
        char filt_loc_auth;
        char filt_rem_auth;
        char **filt_undef_servers;

        /* FTP proxy filters - all start with "pf" */
        char filt_port;         /* filter "pfo" disables PORT command */
        char filt_pasv;         /* filter "pfa" disables PASV command */
        char filt_get;          /* filter "pfg" disables RETR command */
        char filt_put;          /* filter "pfp" disables STOR command */
        char filt_site;         /* filter "pfs" disables SITE command */
        char filt_mkdir;        /* filter "pfm" disables MKD  command */
        char filt_rmdir;        /* filter "pfr" disables RMD  command */
        char filt_delete;       /* filter "pfd" disables DELE command */
        char filt_rename;       /* filter "pfv" disables RNFR & RNTO
commands*/
        char filt_anon;                 /* filter "pff" disables USER ftp and
anonymous */
        u_long filt_size;       /* filter "pfSN" sets N KB to max file size */
} ftp_acl_filter_t;
```

Here are some example filter strings, from acl_load.c:

```
        /* FTP: site, del, WWW: java, activex, cookies */
#define FILTER_STR1     "pfs|pfd|pwj|pwa|pwc|"
        /* generic filter: debug=3   */
#define FILTER_STR2     "pgd3|"
        /* debug=2, FTP: 69K, strong auth, with external auth servers */
#define FILTER_STR3     "pgd2|pfS69|pgaS|pgA:safeword:radius|"
```

SQUID Issues

The caching WWW proxy (squid) is very interesting because it has its own ACL checks and non-blocking DNS interface. We leveraged this built-in support in our work, but it was still tricky to integrate the firewall's ACL calls while operating as a non-blocking long-lived proxy.

Squid supports something called proxy-authentication, but this will only work if somcone has configured their web browser to contact a proxy for all URLs. Before doing ACL checks, we use the following code to handle this special case:

```
if (scc_getregion(&conn-;>me.sin_addr) == 0)
        name_valid = 1;        /* non-transparent mode supports proxy-auth */
else
        name_valid = 0;        /* transparent mode does not */
```

This will cause ACL checks for transparent HTTP requests to bypass user nodes, and squid will ignore auth filters. Non-transparent requests (where the connection is TO the firewall) will enforce any user nodes and auth filters in the ACL tree.

Since the proxy might not get an authentication filter after the ACLs return NEEDS_USERNAME, the squid proxy-auth code has been changed to not return a failure code if the password was not accepted. Instead we save some internal state, and only check this state if an authentication filter is returned later.

It is worth noting that in non-transparent mode squid can proxy and authenticate http, gopher, ftp and wais URLs.

In the Proxy

The proxy will make two calls to the ACLs. The first will be:

```
int scc_is_service_allowed(
        unsigned long service_number,
        struct sockaddr_in *src_ip,
        struct sockaddr_in *dst_ip,
        char *src_host_name,                    /* usually null */
```

```
        char *dst_host_name,            /* usually null */
        char *user_name,                /* null if none */
        int name_valid,                 /* tell if name is valid */
        /* return values */
        int &to_region;
        int &from_region;
        int &filter_text_len,
        char &filter_text,
        int rule_name_len,
        char &rule_name,
        struct sockaddr_in &redirect_src_addr_port,
        struct sockaddr_in &redirect_dst_addr_port,
        int &master_key,
        caddr_t &connection_id          /* id for this connection */
        );
```

The possible return values will be:

```
#define ACL_DENY 0
#define ACL_ALLOW_HIDE_SRC 1
#define ACL_ALLOW_HIDE_DST 2
#define ACL_ALLOW_HIDE_BOTH 3
#define ACL_ALLOW_SHOW_ALL 4
#define ACL_RESOLVE_SRC_ADDR 5
#define ACL_RESOLVE_DST_ADDR 6
#define ACL_NEED_MORE_FILTER_SPACE 7
#define ACL_NEED_USER_NAME 8
```

Thus the ACLs will return, for each connection, how to hide the addresses. The description of each of these values is as follows:

service_number: this is a number that the backend decides and is unique per service or possibly per service, from and to region triplet as desired.

src_ip: this is the source IP address of the connection.

dst_ip: this is the destination IP address of the connection.

src_host_name: this is the host name based on the reverse lookup of the source address of the connection. This is generally only used when the kernel explicitly asks for it by returning from a previous call to *scc_is_service_allowed* with a return value of *ACL_RESOLVE_SRC_ADDR*.

dst_host_name: this is the host name based on the reverse lookup of the destination address of the connection. This is generally only used when the kernel explicitly asks for it by returning from a previous call to *scc_is_service_allowed* with a return value of *ACL_RESOLVE_DST_ADDR*.

user_name: this is the user name of the person using the service. This value is only used when *ACL_NEED_USER_NAME* has been returned by the kernel. Use NULL, if the name has not yet been requested. Currently only FTP, telnet and WWW support user names.

name_valid: this tells the ACLs whether or not a user name makes any sense for this protocol. If the *name_valid* flag is set to TRUE, then user decision nodes will be used (and thus a user name will be required if a user decision node is encountered when checking the ACL). If set to false, then the user decision nodes will be ignored and the true path of those nodes encountered when checking the ACL will be used.

to_region: the region number that the destination address of this connection is in.

from_region: the region number that the source address of this connection is in.

filter_text_len: this is a pointer to an integer which has the length of the *filter_text* array in it. This value will be set to the amount of data returned by the access call on return. If the return value is *ACL_NEED_MORE_FILTER_SPACE*, then the value in this variable will contain the amount of space required.

filter_text: this is an array of characters of size *filter_text_len* which will be used to store the concatenated filter strings accumulated while checking the ACLs.

rule_name_len: this is the size of the array *rule_name*.

rule_name: this is the name of the rule that allowed or denied the connection. Only a maximum of *rule_name_len* - 1 characters will be stored in there.

redirect_dst_addr_port: this is the address and port to redirect this connection to. The system will set this to all zeroes if it is not in use. The port and address will always both be set together in this structure if it is to be used. Only the *sin_port* and *sin_addr* part of the structure will be used.

redirect_src_addr_port: this is used to indicate to the firewall that when making the connection from the firewall to the destination, it should use the source address/port provided. Note that unlike the *redirect_dst_addr_port* field only the parts of the address required will be filled out. In particular, if the port is specified but not the address then the address field will be zero. Similarly, if the address is specified but not the port, then the port will be zero. For the *redirect_dst_addr_port*, if one or both field are specified then they are both returned (with the unspecified field left the same as the actual destination).

master_key: this is the key that indicates which items have been licensed on the firewall.

connection_id: this is the connection id for this connection. When the service is finished you provide this id to the *scc_service_done* system call and that function decrements the correct counters.

Note that the user name will be used by the system to get the groups automatically behind the scenes in the library call. This means that the *actual* call to the kernel will have more fields. In particular, there will be a list of group names and a counter to indicate how many elements are in the list.

The second call will be:

```
int scc_service_done(caddr_t connection_id);
```

This call always returns zero now. The kernel will use the information in the *proc* structure for this process to decrement the connection counts for this connection.

There is one other call that a proxy might have to make. When an ACL is updated, proxies have to recheck their connections to see if they can still make the connection. This is done as follows:

```
int scc_recheck_service(
        unsigned long service_number,
        struct sockaddr_in *src_ip,
        struct sockaddr_in *dst_ip,
        char *src_host_name,            /* usually null */
        char *dst_host_name,            /* usually null */
        char *user_name,                /* null if none */
        int name_valid,                 /* tell if name is valid */
        caddr_t &connection_id          /* id for this connection */
        /* return values */
        int &to_region;
        int &from_region;
        int &filter_text_len,
        char &filter_text,
        int rule_name_len,
        char &rule_name,
        struct sockaddr_in &redirect_src_addr_port,
        struct sockaddr_in &redirect_dst_addr_port,
        int &master_key
        );
```

Returns from this will be the same as for the *scc_is_service_allowed* call except that *connection_id* is passed in as a parameter not a return value.

If the connection is not allowed, then the counters are automatically freed up and the proxy need not make any further calls for that connection. In the case of counter nodes, the recheck will fail until the counter is at an acceptable level. This means that, if the counter has been decreased below current connection levels, the first connection rechecked will fail and so on until the current number of connections counter has been decremented enough. Thus, proxies should recheck services in order of lowest priority to highest priority (typically by checking the oldest sessions first, when that is possible). Note that short-lived proxies and servers started by secured cannot guarantee the order in which ACLs will be rechecked, since they will all get a HUP signal at the same time.

The following new system calls were added to BSDI 3.1 version of UNIX to support regions:

rgnbind()	allows a service on the firewall to listen for network connections only in the specified region. This allows us to have different programs listening in different regions; for example, a caching WWW proxy for connections from internal to external and a non-caching proxy from SSN to external. In one embodiment, network servers were modified to use rgnbind() instead of bind(), to ensure that they handled traffic for the correct region.
rgnctl()	adds, deletes, and modifies regions and sets per-region parameters: Members, router, connection refused, and ping response.
rrctl()	sets region-to-region policy. Currently only handles network address translation, but could add other parameters in future.
scc_getregion()	retrieve the region number for a given IP address
scc_service_checks()	
scc_backend_acl_calls()	
scc_service_done()	
scc_get_service_counts()	

Other changes include:

- initialization of region table at system startup time;
- addition of a region number to the packet header data structure to record the region ID for every network packet received;
- addition of a field to the network interface data to record which region that interface belongs to; and
- addition of a field to the VPN security association data to record which region the VPN is belongs to.
- In the ICMP (Internet Control Message Protocol) processing, if the incoming packet is an ICMP ECHO_REQUEST (commonly known as a "ping"), check the region table and only respond if ping response is enabled for the region from which the packet came;
- In the IPSec key and policy processing code, code was added to record the region ID associated with keys and policy table entries, and to manipulate keys and policies on a region-by-region basis;
- List of changed files: Region modifications were made to the following files within the BSD/OS kernel:

kern/uipc_mbuf.c
kern/uipc_syscalls.c
ACL/aclservice.c
netinet/ip_input.c
netinet/in_pcb.c
netinet/in_pcb.h
netinet/ip_icmp.c
netinet/ip_tunnel.c
netinet/raw_ip.c
netinet/tcp_input.c
netinet/udp_usrreq.c
netkey/key.c
netpolicy/policy.h
netpolicy/pt_debug.c
netpolicy/ptsock.c
netpolicy/policy.c
netsec/ipsec.c
netsec/ipsec_ah.c
netsec/ipsec_esp.c
sys/aclkern.h
sys/audit_codes.h
sys/mbuf.h
sys/region.h
sys/sysctl.h
net/if.c
net/if.h

Region Determination Processing

Referring to Figure 5, when a packet is received as shown in step 80, the region ID is retrieved from the network interface and assigned to the packet in step 82. It is determined in step 84 whether the packet is encrypted, i.e., a VPN. If the packet is encrypted, processing proceeds to step 86 where the VPN security association for that packet is retrieved. The packet is then decrypted in step 88, and the previously stored region ID for that packet is replaced with the region ID of the VPN in step 90. All further operations take place on the decrypted packet.

Ordinarily, a UNIX system then checks whether the packet is destined for one of the firewall's IP addresses. If not, the packet is forwarded to the real destination. This has been modified in SecureOS to check that: (a) the destination is in the same region as the source and (b) the "router" flag is set for that region, as shown in steps 92 and 94. If either condition is not met, the packet is not forwarded, as shown in step 102.

In step 96, the system looks for any socket listening for the incoming packet. Traditionally this match looks at source IP address, source IP port, destination address, and destination port. This has been extended in SecureOS, as shown in step 98, to also check the region associated with the packet against the region specified in the rgnbind() system call, to ensure that sockets receive data originating only from the correct region. If all conditions are met, the packet is forwarded in step 100; otherwise, the packet is not forwarded (step 102).

This folowing example sets up three regions: internal, external, and Secure Server Net (SSN):

Name	Members	We show address to	We see address from	Rtr	Conn	Ping
Internal	ef0 Ethernet		External	1	1	1
	VPN-Waterloo		SSN			
External	ef1 Ethernet	Internal SSN		0	0	0
SSN	ef2 Ethernet	Internal	External	1	1	1

The fields are:

Name	user specified region name
Members	physical interfaces and VPN encrypted connections that belong to this region.
We Show Addr To We See Addr From	the Network Address Translation configuration. This example shows that the Internal region is hidden from all others, and that the SSN region is hidden from External but visible to Internal
Rtr	if 1, the firewall acts as a router between members of this region. In this example, packets would flow between the Internal region and the VPN to Waterloo as if they were simply going through a router.
Conn	If 1, the firewall returns "connection refused" messages if there is no service available on the requested network port. Setting this to 0 on external regions can help defeat network scanning attacks.
Ping	Respond to network pings (ICMP ECHO-REQUEST packets). Again, setting to 0 on external regions can help defeat network scans.

The following example shows a region of the firewall of the present invention configured to sit between two departments of a company and transparently filter and control network access between the departments.

Name	Members	We show address to	We see address from	Rtr	Conn	Ping
Service	ef0 Ethernet	Research	Research	1	1	1
Research	ef1 Ethernet	Service	Service	1	1	1

The two regions can see each others' IP addresses; that is, no address translation is done. Nevertheless, network connections are only allowed if an access rule on the firewall grants permission.

The ACLs described above combine the services themselves, the regions that the services bridge, and the access control decisions. The user draws a graph which starts with a service and a to-from set. Next, the user creates a path consisting of the desired options which can include: time, session counts, authentication, encryption, users/groups, WWW filters, ftp filters, email filters, destination address re-writes, to addresses and from addresses. The user is building a decision tree.

In the embodiment shown, some of the decision nodes in the tree have two paths from them to the next node (a true path and a false path) and some just have one path. The nodes that have one path are nodes which provide filtering, logging, or address rewrites. No decisions are made on filtering since filtering is performed in user level code. (For example, to make the implementation easier, the kernel will not try to implement SmartFilter. Instead, the result of the ACL check will be to provide a response which notes SmartFilter should be applied and supplies the categories which are to be blocked. The proxy will allow the connection provided that the SmartFilter check allows the connection.)

As noted above, in one embodiment each node in the decision tree can be one of two types of node. The first type is a decision node. The second type of node is a *filter* node.

A decision node is one where the decision regarding the action to perform is done in the kernel. To the user, on the GUI, it means that they can have a *true* branch and a *false* branch. This node is implemented in user space in the service itself.

A filter node is implemented in user space in the service itself. The service will ignore filters which do not apply to it. To the user, on the GUI, it means that they can only have a *true* branch. The *false* branch is always a *deny service.*

Decision Nodes

This section describes one embodiment of the decision nodes and their associated data structures. Also described are the system calls that will be available to work on the node. This design assumes that each ACL will consist of first a list of services, followed by some to/from region decisions, and then followed by anything else desired.

The *scc_decision_node* is a *union* structure that looks like this:

```
struct scc_decision_node {
     char *node_descriptor;
     loop_check;
     scc_decision_node *true_path;
     scc_decision_node *false_path;
     int reference_count;
     int node_has_been_deleted;
     int node_type;
     int debug_node;
     union {
          scc_user_rec user_struct;
          scc_addr_rec addr_struct;
          scc_counter_rec counter_struct;
          scc_decision_node *subrule_ptr;
          scc_date_rec date_struct;
          scc_filter_rec filter_struct;
          scc_log_rec log_struct;
          scc_rewrite_rec rewrite_struct;
          scc_mat_rec mat_struct;
     } detail_data;
}
```

node_type is one of:

```
#define ACL_SERVICE_DECISION_NODE 0
#define ACL_USER_DECISION_NODE 1
#define ACL_ADDR_DECISION_NODE 2
#define ACL_COUNTER_DECISION_NODE 3
#define ACL_RULE_DECISION_NODE 4
#define ACL_DATE_DECISION_NODE 5
```

```
#define ACL_FILTER_NODE 6
#define ACL_LOG_NODE 7
#define ACL_REDIRECT_NODE 8
#define ACL_PERMIT_SERVICE 9
#define ACL_DENY_SERVICE 10
```

which describes which of the union pointers to use. And, in the case of the end of the path, the *node_type* indicates if a permit or deny is to be used.

Note that the *subrule_ptr* is to implement the rule within a rule requirement of the GUI.

If a decision to check is *true*, then the *true_path* is the next node to check. Similarly for a *false* decision.

The *node_descriptor* is a character string which describes this particular node. There is no set definition for this description so the backend is free to enumerate nodes as it wishes and the GUI/backend can use node descriptors to glue together messages from the audit stream to trace through what is happening in the decision process. Also we use the node descriptor as an index into a the node table. This table has as entries a pointer to each node for fast node lookup.

If a node is deleted, then the *node_has_been_deleted* flag is set. If at any point in a ACL check we come across such a node we issue a deny. We use the *reference_count* to determine if we actually delete the node. Only when the reference count is zero do we actually free up the memory.

The *debug_node* flag can be set to do various things as will be discussed below.

We use the *loop_check* flag to prevent loops in the ACLs causing us to recurse forever. We set this flag to true when we enter this node for checking and after checking the children to the end we reset the value to false. If while checking the children we encounter a loop flag set to true we know we have reached cycle in the tree.

The Services node and regions node are special decision node which anchor the decision tree. This allows for quick indexing by service number. To do this, there will be an array of pointers (*scc_service_array*) indexed by the service number. The pointers point to and array of regions used by that service. There will be a variable *max_service_number* which the kernel will maintain to

use as a guild line for indexing into the service array. Each entry in the *scc_service_array* will be a structure as follows:

```
struct scc_service_def {
     int number_regions;
     scc_service_rec **region_array;
     int reference_count;
     int node_has_been_deleted;
     }
```

Each service should have a unique number but this will not be implemented in the kernel. Rather, the kernel will be given a service number and the kernel will allocate a bucket for that service. The kernel will be unconcerned about which service this bucket actually belongs to. Note that the *scc_service_rec* is not a part of the *scc_decision_node* listed above.

When we want to delete everything for an entire service (a user defined service for example), we check to see if all structures pointed at by the region array are empty. If not we mark this node as being deleted. The *scc_service_rec*s pointed at by the region array will decrement the reference count as they get deleted (and freed) and when the reference count is zero this record is freed.

```
struct scc_service_rec {
     int number_regions;
     unsigned long total_sessions;
     unsigned long current_sessions;
     scc_decision_node *next_decision;
     char *node_descriptor;
     scc_service_def *parent_struct;
     int to_region;
     int from_region;
     int debug_node;
     int node_has_been_deleted;
     }
```

When an ACL check is requested we use the service number to index into the table of services. This leads us to a *scc_service_def* structure. We index into the *region_array* using the *to* and *from* regions. If the entry has a value (i.e. the pointer is not NULL) and if the *node_has_been_deleted* flag is false, then this

node is the start of a valid decision check. In this case, we use the *scc_decision_node* pointer to start traversing this tree. If no decision tree is found for this particular region-service combination, then the service is denied access.

We will keep track of sessions for each service-region combination so that other programs can check to see the status of traffic on the box. Thus, every time a success is returned, the counts here are increased and every time service de-registers, the proper counter, *current_sessions*, is decreased. The *node_has_been_deleted* is there for when the service record is to be deleted. In this case, processes will continue to decrement the *current_sessions* until all the counts are zero. At that time, the memory will be freed. When we can free this structure we go back to the parent structure and NULL the entry in its *region_array*. If all entries are null then free that structure if it is marked for deletion.

The user decision node is used to make decisions specific to users or groups of users. This structure is simple and goes like this:

```
struct scc_user_rec {
     char **sorted_user_array;
     int number_of_users;
     }
```

If the user being checked is in the array of users, then the decision is *true*. If one of the groups that the user belongs to (also included in the system call) is in the array of users, then the decision is *true*. Note that users and groups are one and the same as far as the system goes. This means the GUI/backend must make sure that there is not a group called *Andrew* **and** a user called *Andrew*.

If no user name was provided for the ACL check and if user names are relevant to this protocol (i.e. the *name_valid* flag was set to true in the ACL call), then if the calling process does not provide a user name,

```
#define ACL_NEED_USER_NAME 8
```

will be returned. The proxy would need to query for a user name and call again with that information.

The IP Addresses/Host Names decision node is used to make decisions that select for/against source or destination addresses or host names.

```
struct scc_addr_rec {
     int type;
     char **sorted_hostname_array;
     int number_of_names;
     radix tree of host numbers/network masks
        }
```

Where type is either:

```
#define ACL_ADDR_SRC_CHECK 0
#define ACL_ADDR_DST_CHECK 1
```

The same structure is used for a source address check and a destination address check. Note that if the address/mask set does not contain the current address being examined and if *sorted_hostname_array* contains some data, then if the correct name was not provided in the ACL call to the kernel, the kernel will return the value:

```
#define ACL_RESOLVE_SRC_ADDR 2
#define ACL_RESOLVE_DST_ADDR 3
```

indicating which address needs to be resolved via a reverse DNS lookup. The ACL would then be called again with the resolved name.

Note that the list of host names must be in sorted order but the letters of the hostname must be **reversed**. For example, *rafael.tor.securecomputing.com* would be *moc.gnitupmoceruces.rot.leafar*. These are then put into sorted order. This allows the kernel to quickly process wild card entries. It is also important that unneeded entries are not loaded into the kernel. For example if the user has specified **.com*, then no other entries of the form *.com* should be present in the list passed to the kernel.

The Maximum Concurrent Sessions decision node provides the ability to put a choke on the number of concurrent sessions on a service or group of services. We want to have the ability to program a counter to be shared among all the services on this path, or to have the counter count for each service individually.

The structure to handle this looks like this:

```
struct scc_count_rec {
     int service_specific_flag;
     unsigned long current_count;
     unsigned long max_count;
     unsigned long total_count;
     scc_detail_count_rec **service_counters;
     unsigned long num_services;
     }
```

Where the *service_specific_flag* can have values:

```
#define ACL_SHARE_COUNT 0
#define ACL_INDIVID_COUNT 1
```

and if the *service_specific_flag* has value *ACL_SHARE_COUNT*, then the shared count record is used. Otherwise, the array is used. Note the size of the array is stored in *num_services* and the array is indexed as:

```
service_counters[service_number].
```

The *scc_detail_count_rec* is:

```
struct scc_detail_count_rec {
     int number_regions;
     unsigned long service_number;
     unsigned long *current_count;
     unsigned long *total_count;
     scc_decision_node *parent_record;
     int node_has_been_deleted;
     }
```

where the current count tells how many connections that use this counter are currently active. The total count is the total number of connections that have used this counter. We use the *max_count* from the *scc_count_rec* to determine the max. Thus the max is a shared value that all individual counts must adhere to. The arrays in the *scc_detail_count_rec* are indexed as:

current_count[to_region][from_region].

Each time a detailed record is allocated the parent decision node's reference count is incremented.

The *node_has_been_deleted* tells a process that is going to decrement the counter whether this node is being used or not. If set to false, then the record is in use and increments or decrements are done accordingly. If set to true, then when the count gets decremented to zero, the memory is freed up and the parent's reference counter is decremented. If the parent has been deleted and if the reference counter is set to zero then the parent node is freed.

The *node_has_been_deleted* flag, in the detailed record, gets set to true (i.e. not zero) when the node itself goes away (the user has removed it from the diagram) or if the counter is switched from individual to shared service counts. Note that each counter is indexed by *to* region and *from* region so that the count is unique on a service-from region-to region triplet.

The *parent_record* pointer points back to the top level *scc_decision_node*. The *service_number* is there so that we can index into the *service_counters* array and set the array pointer to NULL when we are preparing to free up memory.

When the counter is switched from individual to shared service, then the records in the array are all invalidated. The totals of the counts of the array are added up as the new *total_count* for this node (in the parent record). In order to keep the counters correct, when we decrement a counter, check to see if the record has been deleted. If the record is marked as deleted and if the *parent_record* is set to shared, then decrement the shared counter as well. If the individual counter is now zero, free up the memory as above.

The Time/Date decision node provides the ability to use date and time as a means of restricting access to services. The structures look like this:

```
struct scc_date_rec {
     unsigned long number_details;
     scc_date_detail_rec *date_details;
     }

struct scc_date_detail_rec {
     unsigned long start_seconds;
     unsigned long end_seconds;
     }
```

The *scc_date_rec* is the top level structure and it has *number_details* separate date rules. Each of those rules are in a *scc_date_detail_rec*. So, we have an array of structures in *scc_date_rec* each of which has a start seconds and an end seconds value. Each value is relative to the beginning of Sunday. Thus, start second 0 and end second 1 would be allowing the connection only during the first second of Sunday.

The backend must provide the records in sorted order by *start_second*.

A time and date decision is based on a series of time rules. We simply check the current time and day against each rule. If we find a rule where the current time and day falls in that rule, then the decision is the true path otherwise it is the false path.

In one embodiment, to be a complete rule, a rule must consist of at least a services node and a region node and have all true and false branches terminated by terminal nodes. If you plan to use a segment of a particular rule in more than one rule, you can create a partial rule. Partial, or shared, rules can be added to any complete rule.

In one embodiment, complete or partial access rules can be configured using a graphical user interface such as is shown in Fig. 3. In order to configure a complete or partial rule one must perform the following general steps:

1) decide if you want to create a complete or partial rule.

2) select the services this rule will control.

3) select the source region and destination region for the rule.

4) decide on a name for the rule.

5) decide what nodes you want to add to this rule.

The ability to create shared rule segments is built into the system as follows. A rule is simply a chain of decision nodes. After the chain of rules is completed, the decision path at the entry point to the sub-rule is taken based on the outcome of the rule. The filters and audit messages within the rule are still generated and accumulated.

Log nodes direct the kernel to log messages to the audit subsystem. The backend can fully specify the message to log. The structure is as follows:

```
struct scc_log_rec {
     char *audit_message;
     int audit_message_type;
     }
```

The message will be of the form:

```
audit_printf(audit_message_type,
     AUDIT_A_AREA,
     AUDIT_T_NETACL,
     AUDIT_P_MAJOR,
     "%s from ACL log node: %s",
     node_descriptor,
     audit_message);
```

Note that an *scc_log_node* always takes the true path of the decision tree.

In one embodiment, filters are just strings which the proxy interprets to perform it's filtering. The kernel does none of the decision work. Instead, the kernel is given a pattern, and if the node is reached and if there is some data for the decision made at that node, then the pattern is accumulated as a filter. All of the filters are accumulated by the kernel, concatenated together and returned to the proxy as part of the system call. In such an embodiment, the kernel requires no work to implement filters beyond the re-writing of addresses.

A filter structure contains all the relevant filter data. The following shows the data and explains its use:

```
struct scc_filter_rec {
     char *filter_string;
     int filter_string_length;
     }
```

If the *filter_string_length* is zero, then there are no filters otherwise, this filter string is appended to the array passed in, in the ACL call by the service.

The filters are as follows: encryption, authentication,

The encryption filter requires that a connection is encrypted with a certain level of encryption. It will be up to the user level process to verify that the requirements of the filter are met. If the requirements are not met the action is to deny the connection.

The authentication filter requires that a connection is authenticated. One or more possible methods of authentication can be specified. This would only apply to those protocols that allow for a user name as part of the protocol. Currently this would be: *ftp*, *telnet*, and *WWW*.

There are a number of possible WWW Filters. For instance, SmartFilter can be used as described above. In addition, a WWW filter may block Java or ActiveX scripts. In one embodiment the SmartFilter filter can also specify which policy to use (for sites that define multiple policies). These are performed by the caching WWW proxy only. One such embodiment also includes cookie blocking.

Likewise, there are a number of possible FTP Filters. These include filtering on: *GET*, *PUT*, *PASV*, *PORT*, *MKDIR*, *RMDIR*, *RENAME*, *DELETE*, *SITE*, filtering on file size and filtering anonymous ftp. All filtering must be done by the proxy or server.

Furthermore, there are a number of email filters required. This includes mail mapping and content blocking. Again the proxy/server must fulfil the requirements of the filter.

Redirect nodes

Redirect nodes act like filters since they only have one path out of them. Redirects are tables which map source or destination addresses to other source or destination addresses. Currently we only map destination addresses. The most

obvious use of redirects are to map connections coming into the firewall from the insecure side of a NAT region pair to a secure machine. In this case, the connecting host cannot see the hosts behind the firewall. The redirects will map a connection coming to a given firewall address (could be one of many because of MAT) to the desired secure host. The kernel will only accept addresses (the UI can accept names providing it translates them to an address). The tables, whose structure is described below, will contain an entry for each MAT address that applies.

Another use of redirects is to map an address going from a region which can see all the hosts in the destination region. In this case, the redirect has only one entry which maps the address and port to the given address and port.

The final case is one where we might not know which of the above two apply. In that case, all possible MAT addresses might be present and a global rule in the case that the connection is not to the Firewall itself, is also present. This final case happens when you are using a redirect from a rule within a rule.

The structure for the redirect table is as follows:

```
struct scc_rewrite_rec {
     int node_type;
     int no_match_flag;
     int num_table_entries;
     scc_rewrite_rule *rewrite_rules;
     }

struct scc_rewrite_rule {
     struct sockaddr_in check_addr;  /* including port
*/
     struct sockaddr_in netmask;     /* netmask used
for checking */
     struct sockaddr_in new_addr;  /* including port
*/
     }
```

The *node_type* is one of:

```
ACL_SRC_REWRITE_NODE
ACL_DST_REWRITE_NODE
```

Since the number of addresses to check against are minimal, we will leave the addresses in unsorted order.

The redirect mapping goes as follows:

1. See if there is an address/port which matches the current connection. A port number of 0 means any port and an address of 0.0.0.0 means any address.

2. If there is a matching address or port, to rewrite the addresses.

3. If there is no match address then if the *no_match_flag* is set to ACL_REWRITE_NO_MATCH_DENY then deny the connection. Otherwise leave the port and address unchanged.

Note that if there are multiple redirect boxes on the path that allows a successful connection, then the one closest to the check mark has priority. Note also that those rules which do not change a value, I.e. if there is a rule which says for any address map port x to port y, then the address is not considered to be mapped and thus a redirect box further away from the check mark could rewrite that address (but not the port). If there is a rule further away which re maps the address and the port then that rule does not apply.

One embodiment supports netmasks in the kernel. Such an embodiment masks the address to check with the netmask and check to see if it is the same as the *check_addr*. If so (and providing there is a port match) we have a match. Thus the *check_addr* and the netmask must match.

MAT nodes

These are nodes that handle MAT address on a single region interface. The GUI system allows the user to configure different behaviors depending on which address the connection came to the firewall on. To handle this the backend needs to put a MAT node as the node the service points to for those regions that have MATs. For example, if the user enables a service `From "region 1"` to `To "Firewall via address a"`, then a MAT node is needed. We only need MAT nodes for the firewall region provided that MAT has been defined for the firewall in that region.

If an ACL check comes to a MAT node and if the destination address is not found in the list of addresses then the connection is denied.

The structure used is:

```
struct scc_mat_rec {
     int num_mat_addrs;
     struct sockaddr_in *mat_addrs;
     scc_decision_node **next_node;
     };
```

In one embodiment, there is a hash table that stores pointers to the decision nodes. The hash table consists of pointers to linked lists. The string is hashed to a bucket in the table and each bucket is the start of a linked list. A node when added to the table, the table is checked to see if the name is unique by looking at the string in the linked list that the string hashes to. If it is unique, then the node is added to the front of the hash table and if the node is already present, an *EEXIST* error is returned.

The hashing algorithm used is the sum of the characters in the name modulo the size of the table. Currently the table is static in size and is set to *ACL_HASH_TABLE_SIZE* (i.e. 200 buckets).

All initialization is done using the *scc_hash_init* function which is called by *scc_acl_init*. The size of the hash table is stored in *scc_d_node_hash_size* and the table itself is stored in *scc_d_node_table*.

Counters need to be kept consistent (i.e. correct) even when a process that holds a connection dies. There are several ways to do this. The current approach is to use the *proc* structure of the process making the system call. A new field will be added to keep track of each counter used by that process and the number of concurrent uses of the counter. When the process dies, then the *exit1* code in the kernel will go through and clear the counters and free the proc space.

In order to make sure that memory is not freed before a process is finished with it, we have a *node_has_been_deleted* flag. This flag is part of every counter and is set to true (i.e. not zero) if the counter is no longer in use and zero otherwise. If a process decrements a current count to zero and if the flag is set to true, then the memory is freed since no process is using that memory. If

a flag is set to true and the current count is already zero, then the memory is freed up immediately.

The following describes one embodiment of the *proc* structure entry for the counters. First, we have a linked list of counters based on a connection. The entry in the *proc* structure is: scc_ACL_cell *scc_ACL_head; Each cell in this linked list is as follows:

```
struct scc_ACL_cell {
     scc_ACL_cell *next;
     scc_ACL_cell *prev;
     scc_service_rec *service_record;
     int number_of_counters;
     int *counter_type;        /* shared or otherwise
*/
     void **counter_rec;
     }
```

The connection id passed back to the proxy will be the actual pointer to the *scc_ACL_cell*. Thus when the proxy does its free, we can very easily free up the counter space, free the memory, and re-attach the linked list of connection information.

When a process exits, we check the linked list of ACL rules and free up any that are still in use by the process.

When a new process starts up, we set the *scc_ACL_head* to NULL. When a process forks, the child's *scc_ACL_head* is set to NULL.

In The Proxy

The proxy will make two calls to the ACLs. The first call is:

```
int scc_is_service_allowed(
     unsigned long service_number,
     struct sockaddr_in *src_ip,
     struct sockaddr_in *dst_ip,
     char *src_host_name,          /* usually null */
     char *dst_host_name,          /* usually null */
     char *user_name,         /* null if none */
```

```
        int name_valid,          /* tell if name is valid
    */
        /* return values */
        int &to_region;
        int &from_region;
        int &filter_text_len,
        char &filter_text,
        int rule_name_len,
        char &rule_name,
        struct sockaddr_in &redirect_src_addr_port,
        struct sockaddr_in &redirect_dst_addr_port,
        int &master_key,
        caddr_t &connection_id   /* id for this
    connection */
        );
```

The possible return values are:

```
#define ACL_DENY 0
#define ACL_ALLOW_HIDE_SRC 1
#define ACL_ALLOW_HIDE_DST 2
#define ACL_ALLOW_HIDE_BOTH 3
#define ACL_ALLOW_SHOW_ALL 4
#define ACL_RESOLVE_SRC_ADDR 5
#define ACL_RESOLVE_DST_ADDR 6
#define ACL_NEED_MORE_FILTER_SPACE 7
#define ACL_NEED_USER_NAME 8
```

Thus the ACLs will return, for each connection, how to hide the addresses. The description of each of these values is as follows:

service_number:	this is a number that the backend decides and is unique per service or possibly per service, from and to region triplet as desired.
src_ip:	this is the source IP address of the connection.
dst_ip:	this is the destination IP address of the connection.
src_host_name:	this is the host name based on the reverse lookup of the source address of the connection. This is generally only used when the kernel explicitly asks for it by returning from a previous call to *scc_is_service_allowed* with a return value of *ACL_RESOLVE_SRC_ADDR*.

dst_host_name: this is the host name based on the reverse lookup of the destination address of the connection. This is generally only used when the kernel explicitly asks for it by returning from a previous call to *scc_is_service_allowed* with a return value of *ACL_RESOLVE_DST_ADDR*.

user_name: this is the user name of the person using the service. This value is only used when *ACL_NEED_USER_NAME* has been returned by the kernel. Use NULL, if the name has not yet been requested. Currently only FTP, telnet and WWW support user names.

name_valid: this tells the ACLs whether or not a user name makes any sense for this protocol. If the *name_valid* flag is set to TRUE, then user decision nodes will be used (and thus a user name will be required if a user decision node is encountered when checking the ACL). If set to false, then the user decision nodes will be ignored and the true path of those nodes encountered when checking the ACL will be used.

to_region: the region number that the destination address of this connection is in.

from_region: the region number that the source address of this connection is in.

filter_text_len: this is a pointer to an integer which has the length of the *filter_text* array in it. This value will be set to the amount of data returned by the access call on return. If the return value is *ACL_NEED_MORE_FILTER_SPACE*, then the value in this variable will contain the amount of space required.

filter_text: this is an array of characters of size *filter_text_len* which will be used to store the concatenated filter strings accumulated while checking the ACLs.

rule_name_len: this is the size of the array *rule_name*.

rule_name: this is the name of the rule that allowed or denied the connection. Only a maximum of *rule_name_len* - 1 characters will be stored in there.

redirect_dst_addr_port: this is the address and port to redirect this connection to. The system will set this to all zeroes if it is not in use. The port and address will always both be set together in this structure if it is to be used. Only the *sin_port* and *sin_addr* part of the structure will be used.

redirect_src_addr_port: this is used to indicate to the firewall that when making the connection from the firewall to the destination, it should use the source address/port provided. Note that unlike the *redirect_dst_addr_port* field only the parts of the address required will be filled out. In particular, if the port is specified but not the address then the address field will be zero. Similarly, if the address is specified but not the port, then the port will be zero. For the *redirect_dst_addr_port*, if one or both field are specified then they are both returned (with the unspecified field left the same as the actual destination).

master_key: this is the key that indicates which items have been licensed on the firewall.

connection_id: this is the connection id for this connection. When the service is finished you provide this id to the *scc_service_done* system call and that function decrements the correct counters.

Note that the user name will be used by the system to get the groups automatically behind the scenes in the library call. This means that the *actual* call to the kernel will have more fields. In particular, there will be a list of group names and a counter to indicate how many elements are in the list.

The second call will be:

```
int scc_service_done(caddr_t connection_id);
```

This call always returns zero now. The kernel will use the information in the *proc* structure for this process to decrement the connection counts for this connection.

There is one other call that a proxy might have to make. When an ACL is updated, proxies have to recheck their connections to see if they can still make the connection. This is done as follows:

```
int scc_recheck_service(
     unsigned long service_number,
     struct sockaddr_in *src_ip,
     struct sockaddr_in *dst_ip,
     char *src_host_name,          /* usually null */
     char *dst_host_name,          /* usually null */
     char *user_name,         /* null if none */
     int name_valid,          /* tell if name is valid
*/
     caddr_t &connection_id   /* id for this
connection */
     /* return values */
     int &to_region;
     int &from_region;
     int &filter_text_len,
     char &filter_text,
     int rule_name_len,
     char &rule_name,
     struct sockaddr_in &redirect_src_addr_port,
     struct sockaddr_in &redirect_dst_addr_port,
     int &master_key
     );
```

Returns from this will be the same as for the *scc_is_service_allowed* call except that *connection_id* is passed in as a parameter not a return value.

If the connection is not allowed, then the counters are automatically freed up and the proxy need not make any further calls for that connection. In the case of counter nodes, the recheck will fail until the counter is at an acceptable level. This means that, if the counter has been decreased below current connection

levels, the first connection rechecked will fail and so on until the current number of connections counter has been decremented enough. Thus, proxies should recheck services in order of lowest priority to highest priority (typically by checking the oldest sessions first, when that is possible). Note that short-lived proxies and servers started by secured cannot guarantee the order in which ACLs will be rechecked, since they will all get a HUP signal at the same time.

In one embodiment, the backend is able to add, change, delete decision nodes. It also is able to insert new nodes into the tree. In such an embodiment, the following functions are provided to allow this to be done efficiently. All backend calls return 0 for success and -1 for failure. Later, *errno* will be used to determine what went wrong.

The Adding New and Updating Nodes call is used to add or update a node. The same call is used to add a new node or update a node. If the *node_descriptor* is unique, then it is a new node, otherwise update the node. In both cases, the values must all be completely filled out.

```
int scc_set_user_node(
     char *node_descriptor,
     char **sorted_user_array,
     int number_of_users,
     char *true_child_node_descriptor,
     char *false_child_node_descriptor);


int scc_set_host_node(
     char *node_descriptor,
     int type,                    /* src or dst check */
     char **sorted_hostname_array, /* see below */
     int number_of_names,
     struct sockaddr_in *ip_addr,  /* array of structs
of ip addrs */
     struct sockaddr_in *ip_mask,  /* array of structs
of ip masks */
     int number_of_ip,
     char *true_child_node_descriptor,
     char *false_child_node_descriptor);
```

Note that the list of host names must be in sorted order but the letters of the hostname must be **reversed**. For example, *rafael.tor.securecomputing.com*

would be *moc.gnitupmoceruces.rot.leafar*. These are then put into sorted order. This allows the kernel to quickly process wild card entries. It is also important that unneeded entries are not loaded into the kernel. For example if the user has specified **.com*, then no other entries of the form *.com* should be present in the list passed to the kernel.

```
int scc_set_count_node(
     char *node_descriptor,
     int service_specific_flag,     /* share or not */
     int max_count,
     char *true_child_node_descriptor,
     char *false_child_node_descriptor);

int scc_set_time_node(
     char *node_descriptor,
     scc_date_detail_rec *date_entries,
     int number_date_entries,
     char *true_child_node_descriptor,
     char *false_child_node_descriptor);
```

Note that the date records must be in sorted order using *start_seconds* as the key to sort on. Note also that the *date_entries* field is an array of structs.

```
int scc_set_filter_node(
     char *node_descriptor,
     char *filter_string,           /* list of filters
*/
     unsigned long filter_string_length,
     char *child_node_descriptor);

int scc_set_log_node(
     char *node_descriptor,
     int audit_message_type,   /* category of audit
call */
     char *log_message,        /* message to output */
     char *child_node_descriptor);

int scc_set_rewrite_node(
     char *node_descriptor,
     int src_dst_flag,
     int no_match_action,
     int number_of_rewrite_rules,
     scc_rewrite_rule *rewrite_rules,
     char *child_node_descriptor);
```

```
int scc_set_subrule_node(
     char *node_descriptor,
     char *subrule_head_descriptor,  /* start of
subrule */
     char *true_child_node_descriptor,
     char *false_child_node_descriptor, /* NULL for
service_rec */
     );

int scc_set_mat_node(
     int num_mat_addrs,
     struct sockaddr_in *mat_addrs,     /* array of
structs */
     char **node_descriptors);
```

Note that for the *scc_set_mat_node* system call, the two arrays must be in sync (i.e. the first MAT address uses the first decision node in the node descriptors array).

Return values from these are as follows:

EEXIST:	there is already a node with this *node_descriptor* and it is different from the node required for the system call.
ENOMEM:	happens when the kernel is out of memory.
ENOENT:	happens when the node descriptor specified does not exist.
EINVAL:	happens when an invalid argument is provided to a system call. One example is if a NULL *true_child_node_descriptor* is passed in as an argument.

```
int scc_set_service_node(
     unsigned long service_number, /* made up by
backend */
     int to_region,
     int from_region,
     char *node_descriptor,
     int node_debug,
     char *child_node_descriptor);
```

The service nodes are different from the other nodes. The reference is the service number not the node descriptor. The node descriptor is there for audit

purposes and should be the name of the ACL rule. If a debug value is set here then debugging is turned on recursively down the tree.

For all nodes, the descriptor to use for the allow terminating node is the string *_SCC_ALLOW*. For the deny connection terminating node, use the string *_SCC_DENY*.

Linking Nodes

Nodes are linked in the same system call that they are built or updated from. Those nodes which only have one path through them only have one potential node leaving them. A child node can either be, a descriptor of an existing node, the string *_SCC_ALLOW*, or the string *_SCC_DENY*. *_SCC_ALLOW* and *_SCC_DENY* are the accept and deny terminals of the tree respectively and otherwise the child is another *scc_decision_node*.

If the child node desired does not exist the system will return an error.

Deleting Nodes

If you want to delete a node you use:

int scc_delete_node(char *node_descriptor);

for all nodes except service nodes. For service nodes you use:

```
int scc_delete_service(
      int service_number,
      int to_region,
      int from_region);
```

Note that this will mark the node as deleted. You must still rebuild the tree. If an ACL is checked and a deleted node is encountered then the ACL will be denied.

Also, the system will only delete nodes when the reference count to that node is zero. All deleted nodes will be removed from the decision node table when the system call is made though.

If you want to delete the service from all regions, then set the source and destination regions to -1;

Debugging Nodes

You can set the debug value of a node (*debug_node* field in the *scc_decision_node* structure) by ORing bits. The possible values are:

```
#define SCC_ACL_DEBUG_TRUE 0x1
#define SCC_ACL_DEBUG_FALSE 0x2
#define SCC_ACL_DEBUG_TIME 0x4
```

If the *SCC_ACL_DEBUG_TRUE* bit is set, then print a debug message when a true decision is reached at this node of the form:

```
audit_printf(
      AUDIT_F_KERN_ACL,
      AUDIT_A_AREA,
      AUDIT_T_DEBUG,
      AUDIT_P_MINOR,
      "ACL node: %s returned true.",
      d_node->node_descriptor);
```

If the *SCC_ACL_DEBUG_FALSE* bit is set, then print a debug message when a false decision is reached at this node of the form:

```
audit_printf(
      AUDIT_F_KERN_ACL,
      AUDIT_A_AREA,
      AUDIT_T_DEBUG,
      AUDIT_P_MINOR,
      "ACL node: %s returned false.",
      d_node->node_descriptor);
```

If the *SCC_ACL_DEBUG_TIME* bit is set, then print a debug message telling how much time was spent in this node in the form:

```
audit_printf(
      AUDIT_F_KERN_ACL,
      AUDIT_A_AREA,
      AUDIT_T_DEBUG,
      AUDIT_P_MINOR,
      "\n\nNode \"%s\" took %ld seconds and %ld
microseconds\n"
```

```
     "Children took %ld seconds and %ld
microseconds.\n",
     d_node->node_descriptor,
     end.tv_sec, end.tv_usec, end_sub.tv_sec,
end_sub.tv_usec);
```

This will include all the time spent in subnodes as well.

You can set the debugging value of a node using a separate system call:

```
int scc_set_debug(
     char *node_descriptor,
     int debug_value);
```

For service nodes you should set the debug value in the *set* system call. Use the same possible values as above.

Service Usage Statistics

In one embodiment, the ACLs keep track of service counts for all services that use them. The counts are by service number, from region, to region triplet. Because we do not know before hand how many services there will be we implement this function in a two call method. A system call which could be used is as follows:

```
int scc_get_service_counts(
     int calltype
     int *count_size,
     struct scc_serv_count *counts);
```

The *calltype* can be one of:

```
#define SCC_GET_NUM 0
#define SCC_GET_VALS 1
```

When called with `calltype =3D=3D SCC_GET_NUM`, this system call sets the value of *count_size* to be the number of elements that need to be allocated in the *counts* array;

When called with `calltype =3D=3D SCC_GET_VALS`, this system call sets the entries in the *counts* array to the appropriate values. If for some reason the number of elements in the array *counts*, passed in *count_size* is not big enough, then the call returns with ENOSPC and passes the new number required back in *count_size*. Even if there is enough space, we return in *count_size* the number of array elements used.

Each entry in the *counts* array is defined as follows:

```
typedef struct {
     unsigned long serv;           /* service number
*/
     int from;                  /* from region */
     int to;                       /* to region */
     unsigned long total_sessions;  /* since last
reboot */
     unsigned long current_sessions; /* current active
*/
} scc_serv_count;
```

Conclusion

It is understood that the above description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reviewing the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

What is claimed is:

1. A method of implementing a security policy, comprising the steps of:

providing a plurality of access policies;

defining a process; and

connecting the access policies and the process to form a security policy.

2. The method according to claim 1, wherein the step of defining a process includes the step of creating an alert.

3. The method according to claim 1, wherein the step of defining a process includes the step of adding a filter.

4. The method according to claim 3, wherein the filter is a URL blocking filter.

5. In a computer network having a plurality of separate networks, an access control mechanism comprising:

a plurality of regions, including a first and a second region;

one or more services bridging said first and second region;

access control rules which define a security policy, wherein the access control rules limit data transfer by the one or more services bridging the first and second regions, wherein the access control rules are defined as a decision tree, wherein the decision tree includes a decision node and a first and a second branch and wherein the decision node includes a true and a false destination path, wherein the true destination path leads to the first branch and the false destination path leads to the second branch; and

access control logic, wherein the access control logic operates with the access control rules to enforce the security policy.

6. The access control mechanism according to claim 5, wherein the first and the second branches lead to other decision nodes.

7. The access control mechanism according to claim 5, wherein the decision tree further includes a filter, wherein the first branch leads to the filter.

8. The access control mechanism according to claim 5, wherein the first and the second branches lead to other decision nodes.

9. In a computer network system having a plurality of networks and a plurality of services, including a first service, wherein each service defines a protocol for transferring data between two of the plurality of networks, a method of limiting transfers between networks, comprising the steps of:

defining a to-from set, wherein the to-from set lists a source network and a destination network;

associating the to-from set with the first service;

defining a path, wherein the path includes desired options for limiting transfer from the source network to the destination network via the first service;

storing information regarding the to-from set, the first service and the path as an access control rule;

receiving a request to set up said first service between the source network and the destination network;

comparing the request to the access control rule to determine access; and

if access is allowed, establishing the service between the source and destination networks.

10. In a computer network system having a plurality of networks and a plurality of services, including a first service, wherein each service defines a protocol for transferring data between two of the plurality of networks, a method of defining a security policy, comprising the steps of:

defining a plurality of access policies, including a first and a second access policy, wherein the step of defining includes the step of creating a plurality of access policy routines, including a first and a second access policy routine, wherein the first access policy routine embodies the first access policy and wherein the second access policy routine embodies the second access policy;

forming a decision tree having a plurality of decision nodes, including a first, second and third decision node, wherein the first and second decision nodes enforce the first access policy and wherein the third decision node enforces the second access policy; and

compiling a list of access control rules, wherein the step of compiling includes the step of replacing each decision node with one of the plurality of access policy routines.

11. In a computer network system having a plurality of networks and a plurality of services, including a first service, wherein each service defines a protocol for transferring data between two of the plurality of networks, a method of enforcing a security policy, comprising the steps of:

defining a plurality of regions, including a first and a second region;

assigning each network to a region;

defining a first and a second service;

defining a plurality of access policies, including a first and a second access policy, wherein the first access policy limits communication between the first and second region using the first service and wherein the second access policy limits communication between the first and second region using the second service, wherein the step of defining includes the step of creating a plurality of access policy routines, including a first and a second access policy routine, wherein the first access policy routine embodies the first access policy and wherein the second access policy routine embodies the second access policy;

forming a decision tree having a plurality of decision nodes, including a first, second and third decision node, wherein the first and second decision nodes enforce the first access policy and wherein the third decision node enforces the second access policy;

compiling a list of access control rules, wherein the step of compiling includes the step of replacing each decision node with one of the plurality of access policy routines;

receiving a packet from the first region; and

accessing the list of access control rules to determine if the packet should be forwarded to the second region.

12. A method of achieving network separation within a computing system having a plurality of network interfaces, the method comprising the steps of:

defining a plurality of regions;

configuring a set of policies for each of the plurality of regions;

assigning each of the plurality of network interfaces to only one of the plurality of regions, wherein at least one of the plurality of network interfaces is assigned to a particular region; and

restricting communication to and from each of the plurality of network interfaces in accordance with the set of policies configured for the one of the plurality of regions to which the one of the plurality of network interfaces has been assigned.

13. A secure server, comprising:

an operating system kernel;

a plurality of network interfaces which communicate with the operating system kernel; and

a plurality of regions, wherein a set of policies have been configured for each of the plurality of regions;

wherein each of the plurality of network interfaces is assigned to only one of the plurality of regions;

wherein at least one of the plurality of network interfaces is assigned to a particular region; and

wherein communication to and from each of the plurality of network interfaces is restricted in accordance with the set of policies configured for the one of the plurality of regions to which the one of the plurality of network interfaces has been assigned.

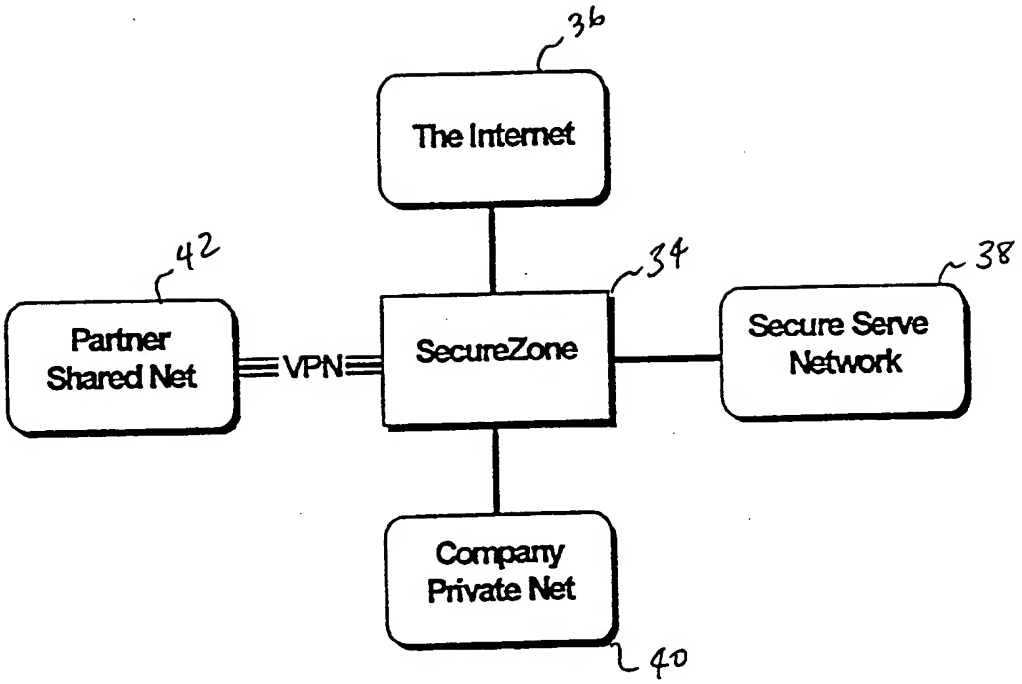

36
The Internet
42
34
38
Partner Shared Net
VPN
SecureZone
Secure Serve Network
Company Private Net
40

Figure 1

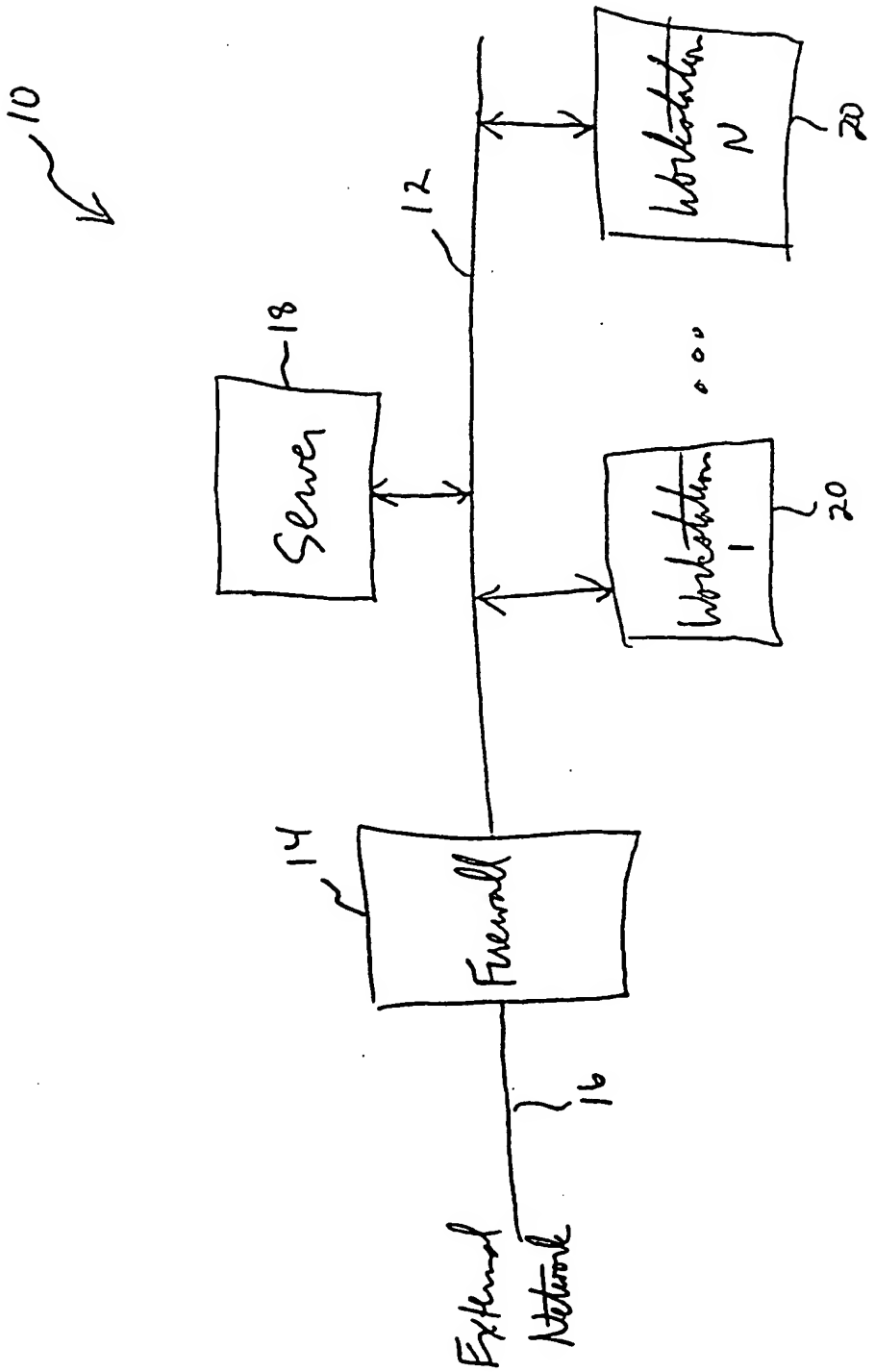

10
12
Workstation N
20
18
Server
Workstation 1
20
14
Firewall
16
External Network

Fig. 1a

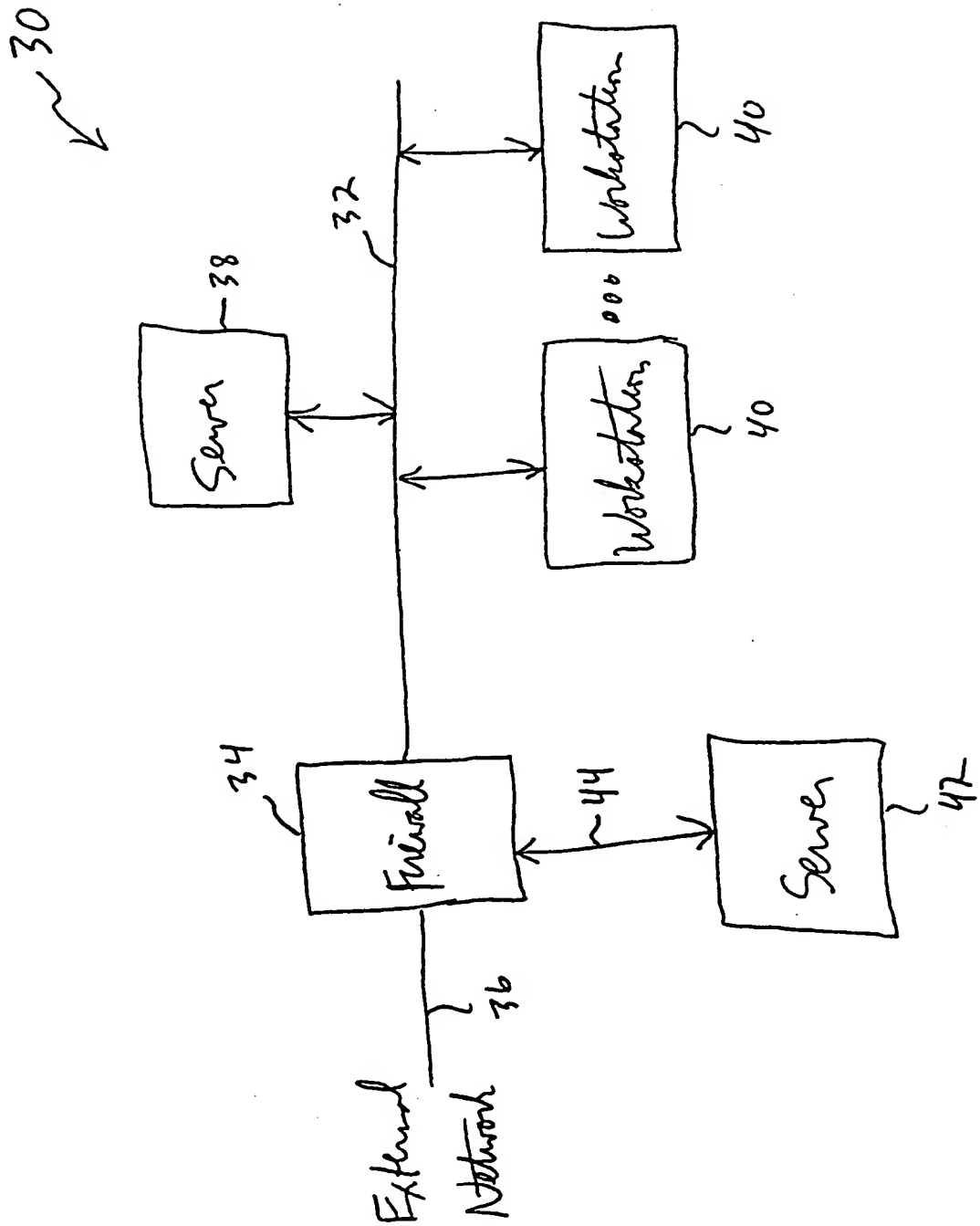

30
32
38
Server
Workstation
40
Workstation
40
34
Firewall
44
Server
42
36
External Network

Fig. 1b

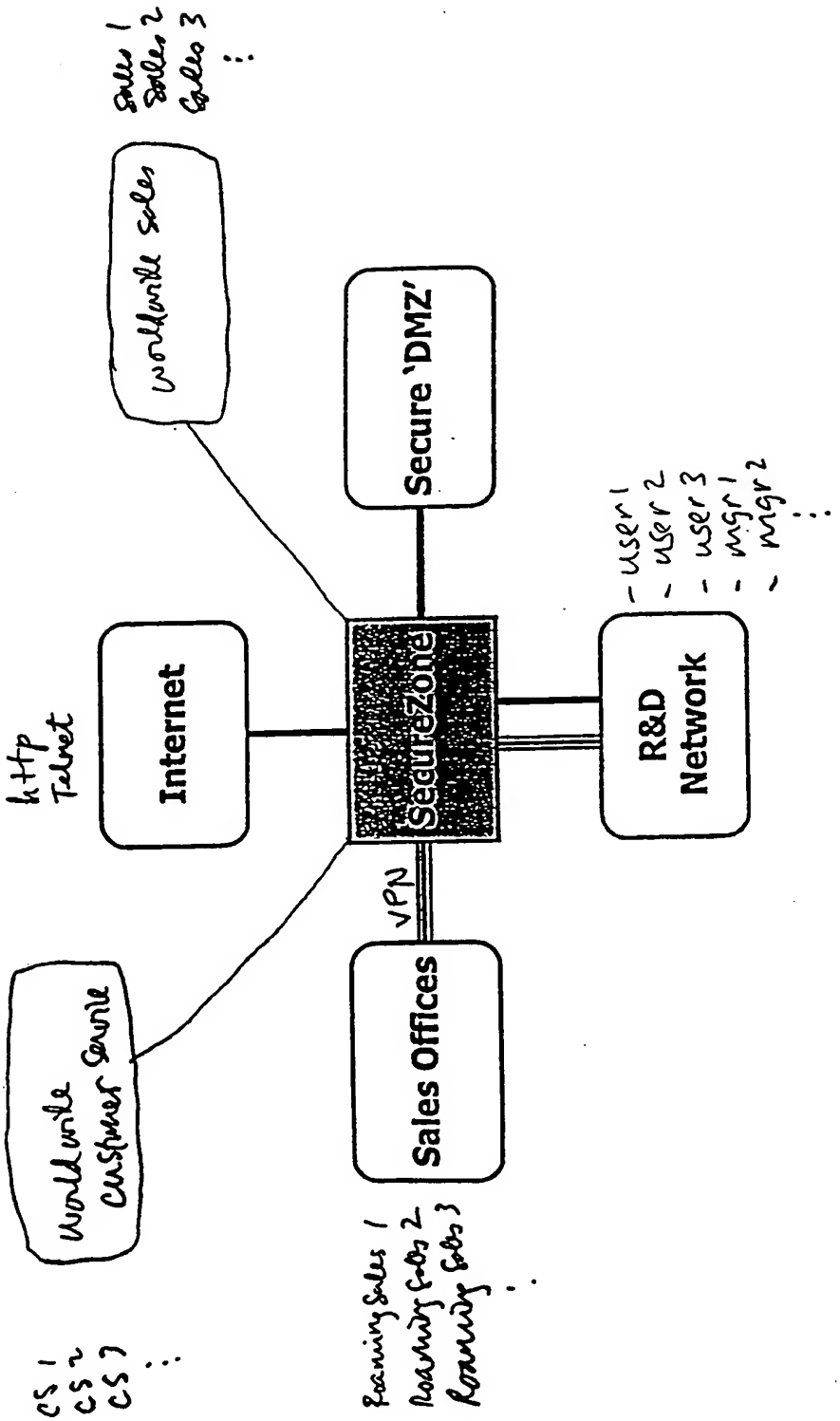

Sales 1
Sales 2
Sales 3
...
Worldwide Sales
Secure 'DMZ'
- user 1
- user 2
- user 3
- mgr 1
- mgr 2
...
http
Telnet
Internet
SecureZone
R&D Network
VPN
Sales Offices
CS 1
CS 2
CS 3
...
Worldwide Customer Service
Roaming Sales 1
Roaming Sales 2
Roaming Sales 3
...

Figure 2

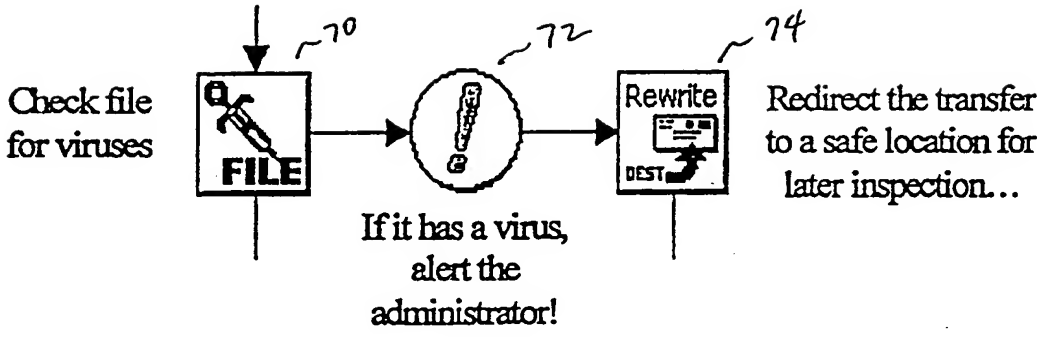

70
72
74
Check file for viruses
FILE
Rewrite
DEST
Redirect the transfer to a safe location for later inspection...
If it has a virus, alert the administrator!

Figure 4

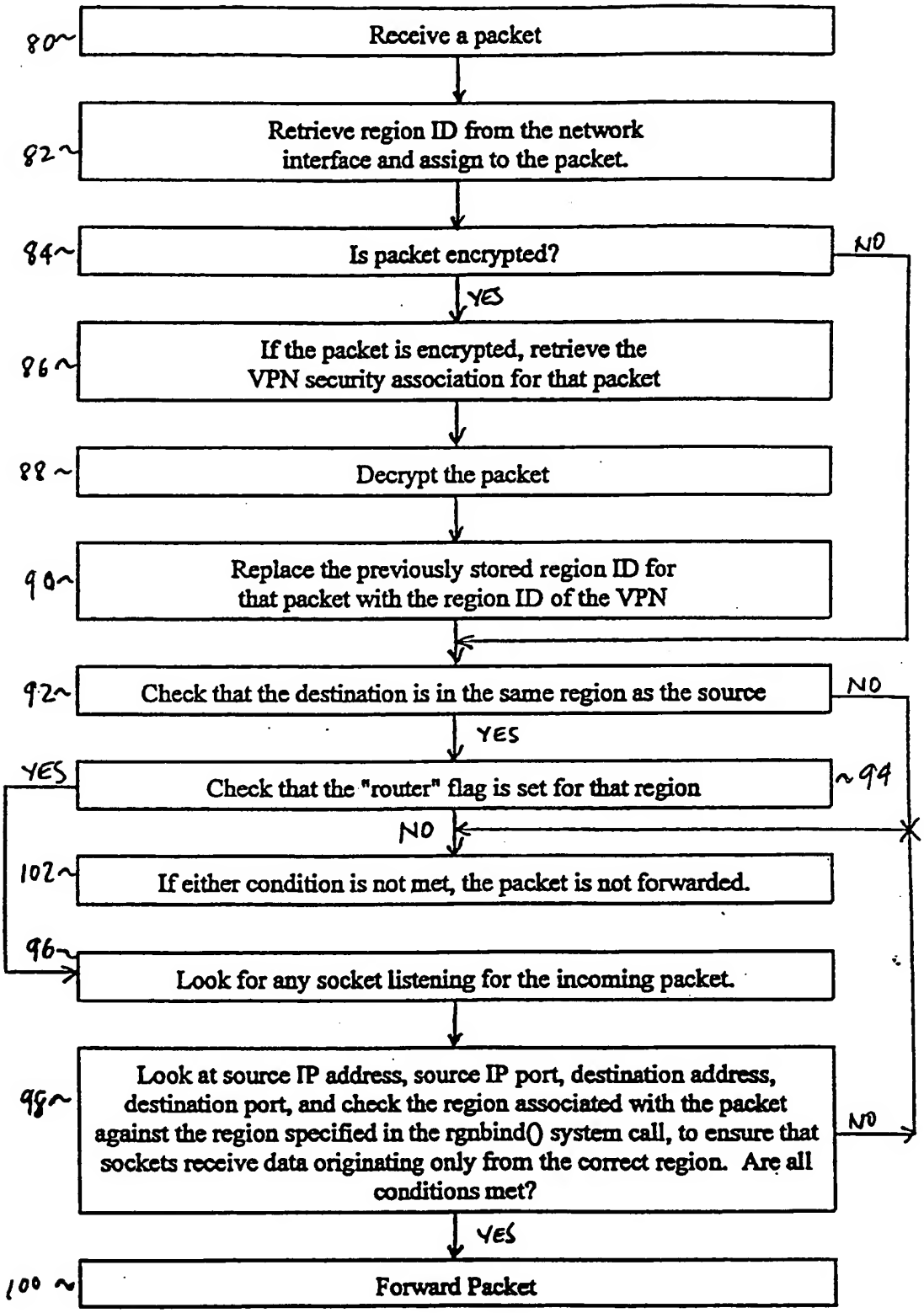

80
Receive a packet
82
Retrieve region ID from the network interface and assign to the packet.
84
Is packet encrypted?
NO
YES
86
If the packet is encrypted, retrieve the VPN security association for that packet
88
Decrypt the packet
90
Replace the previously stored region ID for that packet with the region ID of the VPN
92
Check that the destination is in the same region as the source
NO
YES
YES
Check that the "router" flag is set for that region
94
NO
102
If either condition is not met, the packet is not forwarded.
96
Look for any socket listening for the incoming packet.
98
Look at source IP address, source IP port, destination address, destination port, and check the region associated with the packet against the region specified in the rgnbind() system call, to ensure that sockets receive data originating only from the correct region. Are all conditions met?
NO
YES
100
Forward Packet

Figure 5